View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by PolyU Institutional Repository

26th IEEE International Symposium on Reliable Distributed Systems

The Eventual Clusterer Oracle and Its Application to Consensus in MANETs

Weigang Wu[1], Jiannong Cao[1], Michel Raynal[2]
1. Department of Computing, The Hong Kong Polytechnic University, Kowloon, Hong Kong
2. IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France
{cswgwu, csjcao}@comp.polyu.edu.hk, raynal@irisa.fr

Abstract

This paper studies the design of hierarchical consensus protocols for mobile ad hoc networks. A two-layer hierarchy is imposed on the mobile hosts by grouping them into clusters, each with a clusterhead. The messages from and to the hosts in the same cluster are merged/unmerged by the clusterhead so as to reduce the message cost and improve the scalability. We adopt a modular method in the design, separating clustering from achieving consensus using the clusters. The clustering function, named eventual clusterer (denoted as ◊C), is designed to construct a cluster-based hierarchy over the mobile hosts in the network. Since ◊C provides the fault tolerant clustering function transparently, it can be used as a new oracle (i.e. an abstract tool to provide some kind of information about the state of the system) for the design of hierarchical consensus protocols. Based on ◊C, we design a new consensus protocol, which can significantly reduce the message cost of achieving consensu. We also propose an implementation of the ◊C oracle based on the failure detector ◊S.

1. Introduction

Consensus is a fundamental problem for many distributed computing applications, e.g. atomic commitment, atomic broadcast, and file replication [15][16][17]. Broadly speaking, the consensus problem involves getting a group of processes to agree on a value proposed by one or more of the processes [8][21]. In a distributed system, especially a mobile network, processes are prone to failures. A process is said to be correct if it behaves according to an agreed specification in a run of a consensus protocol; otherwise, a failure occurs and the process is said to be faulty. Precisely, a correct solution to a consensus problem should have the following three correctness properties:

i) *Termination*: Every correct process eventually decides upon some value;
ii) *Agreement*: All the decision values are equal;
iii) *Validity*: Any decision value should have been proposed by at least one process.

Unfortunately, it has been proved that, in asynchronous distributed systems, the consensus problem cannot be solved deterministically even with only one process crash [13]. To overcome this impossibility, several oracles have been proposed, including the *random number generator* [3], the *leader oracle* [5] and the *unreliable failure detector* (FD for short) [5]. An oracle is an abstract tool to provide some kind of information about the state of the system. Based on these oracles, many consensus protocols have been proposed [4][5][9][10][17][18][19] [20][22][24].

The characteristics of wireless networks in terms of communication, mobility and resource constraints introduce new challenges in designing consensus protocols for mobile environments [14][23]. Among others, how to reduce message cost is an important issue, because fewer messages consume less bandwidth, power, and computation resources. In *infrastructured mobile networks*, the message cost can be reduced by shifting the workload of achieving consensus from mobile hosts[1] (MHs) to mobile support stations [1][25].

However, in *Mobile Ad Hoc Networks* (MANETs), there is no mobile support station and the hosts interact with one another in a peer-to-peer way, so the approach used in infrastructured networks is no longer applicable. The hierarchical approach has been widely used in MANETs to achieve message efficiency, stability and scalability. However, how to make use of such a hierarchy in achieving consensus is still a challenging work.

In this paper, we proposed a modular design of hierarchical consensus protocols for MANETs by making use of a cluster-based hierarchy. The hosts are

[1] In this paper, the terms "process" and "host" are used interchangeably.

1060-9857/07 $25.00 © 2007 IEEE
DOI 10.1109/SRDS.2007.24

IEEE computer society

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on March 16, 2009 at 00:09 from IEEE Xplore. Restrictions apply.

grouped into clusters and in each cluster there is a host acting as the clusterhead. The messages from and to the hosts in the same cluster are merged/unmerged and forwarded by the clusterhead so as to reduce the message cost. We separate clustering hosts from achieving consensus using clusters. The function of clustering the hosts is defined as a new object, called *eventual clusterer* (denoted as $\Diamond C$), while the function of achieving consensus using clusters is realized as a consensus protocol HCD (Hierarchical Consensus with Dynamic clusterhead set). HCD and $\Diamond C$ are transparent to each other and executed separately.

The new object $\Diamond C$ is the basis of HCD, which has dual responsibilities of detecting the failures of MHs and constructing a cluster-based two-layer hierarchy over the MHs. Upon receiving a query from a host m, $\Diamond C$ returns three outputs: the set of trusted hosts, the set of the hosts that are acting as clusterheads, and the local clusterhead of m. As $\Diamond C$ requires some additional assumptions for implementation, we call it a new oracle in the following.

The definition and implementation of $\Diamond C$ involve several issues. The core issue in defining $\Diamond C$ is to identify its properties with respect to the requirements of the consensus protocol. To implement $\Diamond C$, we need to consider carefully how to guarantee these properties based on $\Diamond S$, the weakest and commonly used FD [5]. In this paper, following the approach of reducing $\Diamond S$ to $\Diamond W$ [5][7], we propose an implementation of $\Diamond C$. The major challenge in the implementation is how to dynamically select clusterheads.

Like the other oracles, $\Diamond C$ is a transparent basic block that can be used to design new consensus protocols. However, $\Diamond C$ is more powerful than other oracles in the sense that it can be used by the consensus protocols built on top of it to improve their message efficiency and scalability, which is especially important for large scale MANETs. More importantly, the separation of clustering hosts and achieving consensus facilitates the modular design of consensus protocols as the proposal of other oracles. With the help of $\Diamond C$, people can focus on the procedure of achieving consensus using the hierarchy, without worrying about how to establish the hierarchy.

In designing the HCD protocol based on $\Diamond C$, two key issues are addressed. First, the clusterheads are used for not only simply forwarding messages, but also synchronizing the cluster members for the message exchange steps in achieving consensus. Due to the mobility and the failure of the clusterheads, an MH may need to switch between clusterheads that are executing different steps. After switching to a new cluster, the MH has to synchronize with its new clusterhead. Second, nearly all consensus protocols require that no message can be lost. However, the dynamics of the two-layer hierarchy may cause message losses even if the communication channel is reliable and, consequently, may cause MHs to be blocked forever.

How to handle cluster switching is at the core of solving the first issue in HCD. Since the function of clustering (i.e. $\Diamond C$) is transparent to the function of achieving consensus, the latter does not know the detailed information about the implementation of $\Diamond C$ (e.g. whether the host is in the procedure of switching its cluster). Therefore, the function of achieving consensus must be able to keep synchronization between cluster members and the clusterhead without involving the implementation information about the clustering function. In HCD, when a host receives a message from some future round (called "future message"), it will give up its current round and jump to the round of the future message so as to keep synchronization with its clusterhead.

The solution to the second issue involves two aspects. On one hand, when a host finds that its clusterhead has been changed, it sends some redeeming messages to help recover the messages possibly lost due to a crashed clusterhead. On the other hand, some special messages are used to wake up the possible blocked hosts and lead them to a new round.

The rest of the paper is organized as follows. In Section 2, we briefly review existing consensus protocols for mobile computing environments. Section 3 presents the definition and an implementation of the eventual clusterer $\Diamond C$. The proposed HCD consensus protocol using the oracle $\Diamond C$ is presented in Section 4. In Section 5, we analyze the performance of HCD and compare it with similar protocols. Finally, Section 6 concludes this paper.

2. Related work

Several consensus protocols have been proposed for mobile computing environments. Based on the CT protocol [5], Badache et al. [1] proposed the BHM protocol for infrastructured mobile networks. The basic idea of BHM is to let the MSSs achieve consensus on behalf of the MHs. MSSs collect the initial proposal values from their local hosts, and execute the CT protocol to make the decision on some value collected. After the MSSs achieve consensus, they propagate the decision value to all the MHs. A simple handoff mechanism is used to handle the movements of MHs.

The work in [25] extends the BHM protocol by considering the dynamics of the set of MSSs. Using a group membership protocol, the MSSs of the empty cells are deleted from the set of MSSs executing the consensus protocol. Since the group membership

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on March 16, 2009 at 00:09 from IEEE Xplore. Restrictions apply.

problem can also be solved using a consensus protocol [17], two consensus protocols can be involved and executed concurrently. Both the solutions in [1] and [25] rely on the help of MSSs. The principle is to shift the workload from the MHs to the MSSs. In MANETs, however, there is no MSS and all the work has to be done by MHs themselves.

Chockler et al. [6] developed a partition-based consensus protocol for MANETs. The network is divided into non-overlapping grids, each of which is a single-hop sub-network. First, the single-hop consensus is achieved within each grid. Then, each host gossips its single-hop consensus value to the whole network. A host can decide after it has received a value from every grid. Another consensus protocol for MANETs is reported in [28], where several fault tolerant broadcast algorithms for MANETs are designed and applied to a random number generator based consensus protocol for fixed wired networks [12]. Both the protocols in [6] and [28] are probabilistic with respect to their approaches of achieving a global consensus in MANETs.

In our previous work, we have proposed a deterministic consensus protocol for MANETs [29], called HCS (Hierarchical Consensus with Static clusterhead set). Similar to HCD, HCS makes use of a cluster-based hierarchy to achieve message efficiency and scalability. However, HCS has three problems. First, in HCS, the function of achieving consensus is tightly coupled with the function of clustering. When an MH switches to a new cluster, the operations of the consensus function have to be changed according to the execution status of the new clusterhead. Such an approach makes the design of consensus protocol complicated. Second, the set of clusterhead hosts in HCS is static and predefined, so it cannot adapt to the change of the system state, e.g. the crash of some clusterhead, which may delay the decision making. Finally, HCS requires the failure detector of $\Diamond P$ rather than the commonly used $\Diamond S$. The accuracy property of $\Diamond P$ is stronger than $\Diamond S$, which may take a longer time and higher message cost to be satisfied.

The work presented in this paper addresses all the above three problems. Its main contribution is the modular approach to designing hierarchical consensus protocols for MANETs. The function of constructing the cluster hierarchy and the function of achieving consensus are separated and are transparent to each other. The clustering function is defined as a new oracle $\Diamond C$, which can establish a two-layer hierarchy with dynamically selected clusterheads. Since $\Diamond C$ can be implemented using $\Diamond S$, it is equivalent to $\Diamond S$ in terms of the power of failure detection.

3. The eventual clusterer $\Diamond C$

In this section, we describe the eventual clusterer oracle $\Diamond C$. We first introduce the system model and definition of $\Diamond C$. Then, we present an implementation of $\Diamond C$.

3.1. System model

We consider an asynchronous MANET system consisting of a set of n (n>1) MHs, $M = \{m_1, m_2, ..., m_n\}$. An MH can only fail by crashing, i.e. prematurely halting, but it acts correctly until it possibly crashes. There is at least one correct host in the system. MHs communicate by sending and receiving messages. Every pair of MHs is connected by a reliable channel that does not create, duplicate, alter, or lose messages. Of course, in a practical network, transmitting message losses may occur. To cope with such message losses in wireless environments, efforts have been made to design reliable communication protocols [11][27][30]. Since how to guarantee the reliability of the communication channel is out of the scope of this paper, same as in most consensus protocols, we base our work on reliable communication channels.

3.2. The definition of $\Diamond C$

Like unreliable failure detectors or other oracles, the eventual clusterer oracle $\Diamond C$ is also a tool that provides some kind of information about the system. $\Diamond C$ establishes a two-layer hierarchy by grouping the hosts of a MANET into clusters, each of which is managed by a clusterhead. Each host is associated with an eventual clusterer oracle module. On the query from a host m_i, the clusterer oracle module returns three outputs:

i) $\Diamond C.CH$: the set of MHs that currently act as clusterheads;
ii) $\Diamond C.trusted$: the set of hosts that are currently trusted by $\Diamond C$;
iii) $\Diamond C.clusterhead$: the local clusterhead of m_i, i.e. the clusterhead that m_i currently associates with.

Similar to the definition of unreliable failure detectors [5], we define the eventual clusterer oracle $\Diamond C$ using abstract properties.

- *Completeness*: There is a time after which some correct host is permanently included in the clusterhead set $\Diamond C.CH$ and the trust set $\Diamond C.trusted$ at each correct host.
- *Accuracy*: Eventually every host that crashes is permanently excluded from the clusterhead set $\Diamond C.CH$ and the trust set $\Diamond C.trusted$ at each correct host.

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on March 16, 2009 at 00:09 from IEEE Xplore. Restrictions apply.

- *Uniformity*: Eventually, all the correct hosts permanently keep the same clusterhead set $\Diamond C.CH$.
- *Stability*: There is a time after which each correct host is associated with some correct clusterhead permanently.

The *completeness* and *accuracy* properties have been defined for FDs [5]. Here, we used the same names but give different meanings. The completeness (accuracy) in [5] is named accuracy (completeness) here. This is because that the properties in [5] are defined for the set of suspected hosts but here they are defined for the set of trusted hosts.

From the above definition of $\Diamond C$ we can see that, like other oracles, there is a *Global Stabilization Time* (*GST*) for $\Diamond C$ to reach a stable state, i.e. the *stability* property is satisfied. Before *GST*, different hosts may have different $\Diamond C.CH$ sets and a host may switch to a new cluster from time to time. However, after *GST*, all the correct hosts have the same $\Diamond C.CH$ and each correct host associates with a correct host in $\Diamond C.CH$ set. We call such a set the "stable clusterhead set". It is important to notice that a stable clusterhead set includes only correct hosts (but may not be all the correct hosts).

Same as other oracles, $\Diamond C$ facilitates the design of consensus protocols by separating the function of detecting the status of the system and the function of achieving consensus. However, $\Diamond C$ is more powerful in the sense that it can help the consensus protocols built on top of it improve their performance. The messages from and to the hosts in the same cluster are merged/unmerged by the clusterhead so as to reduce the message cost and improve the scalability, which is especially important for large scale MANETs.

3.3. An implementation of $\Diamond C$

To implement a $\Diamond C$, there are two main issues to be addressed: a) failure detection, i.e. the construction of $\Diamond C.trusted$, and b) the construction of clusters. The failure detection can be realized by using FDs proposed in [5]. The failure detector $\Diamond S$ has the following properties:

- *Strong Completeness:* Eventually, every crashed host is permanently suspected by every correct host.
- *Eventually Weak Accuracy*: There is a time after which some correct host is never suspected by any correct host.

Comparing the properties of $\Diamond C$ and $\Diamond S$, we know that the completeness and accuracy of $\Diamond C.trusted$ are the same as the accuracy and completeness of $\Diamond S$ respectively. Therefore, we adopt the unreliable failure detector $\Diamond S$ to detect the failures of MHs.

The second issue can be further divided into two problems: i) the selection of clusterheads, i.e. the construction of $\Diamond C.CH$, and ii) the establishment and maintenance of clusters. Analyzing the properties of $\Diamond C$, we know that the only difference between $\Diamond C.trusted$ and $\Diamond C.CH$ is the uniformity. To establish $\Diamond C.CH$ based on $\Diamond C.trusted$, we adopt the flush algorithm [7] proposed to reduce a leader oracle to the FD of $\Diamond W$, which is equivalent to $\Diamond S$. The corresponding pseudocode is shown as Task c1 in Fig. 1. The code is simple and self-explanatory. In Fig. 1, we use "*CH*" and "*clusterhead*" rather than "$\Diamond C.CH$" and "$\Diamond C.clusterhead$" to refer to the clusterhead set and the local clusterhead respectively.

The pseudocode for the establishment and maintenance of clusters is shown as Task c2 in Fig. 1. First, we show how to construct clusters based on the clusterhead set $\Diamond C.CH$. The clustering procedure is initiated by cluster members. Each host in $\Diamond C.CH$ acts as a clusterhead and manages the corresponding cluster. A host m_i selects the nearest host m_n in $\Diamond C.CH$ using the function *NEAR*($\Diamond C.CH$) and sends a JOIN message to m_n to join the corresponding cluster. On the reception of the JOIN message, if m_n is the clusterhead of itself, it accepts the request of m_i and sends a positive ACK message; otherwise, it rejects the request of m_i and sends a negative ACK message. If a positive ACK is received, m_i ends the switch procedure; otherwise it selects another candidate and repeats the above joining operations. Since $\Diamond C.CH$ is set to M at the beginning, each MH selects itself as the clusterhead and gets the positive ACK when the algorithm starts to execute. Thus, initially, the clusters can be constructed easily.

Now, let us consider the maintenance of the clusters. Due to the host failures or false suspicions, a clusterhead may be removed from the $\Diamond C.CH$ set. Upon detecting the deletion of the local clusterhead from $\Diamond C.CH$ or receiving a RELEASE message from the local clusterhead, a cluster member host m_i will switch to a new cluster. If m_i itself is the clusterhead being moved, it sends a RELEASE message to inform the cluster members; otherwise, it sends a LEAVE message to its current clusterhead. Then, m_i selects the nearest host in $\Diamond C.CH$ as the candidate of the new clusterhead and sends a JOIN message to the candidate. If a positive ACK is received from the candidate, the switch successes; otherwise, m_i selects a new candidate and sends it a JOIN message again. m_i repeats doing this until it is accepted by some clusterhead.

Since we do not assume that the communication channel is FIFO, a sequence number is attached to each

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on March 16, 2009 at 00:09 from IEEE Xplore. Restrictions apply.

```
COBEGIN// The code executed by a host, m_i
----------Task c1: Construction of Clusterhead Set CH -------
(c01) CH ← M; seq_i ← 0;  //M is the set of all MHs,
                          //seq_i is a sequence number;
      ----------Task c1.1: Send CH ----------------------------
(c02) while(true){
(c03)     CH ← CH ∩ ◊S.trusted;
(c04)     send (CH, seq_i) to M;}
      ----------Task c1.2: Receive CH -------------------------
(c05) upon reception of (CH', seq_q) from host m_q:
(c06)    if (seq_q= seq_i) CH ← CH ∩ CH';
(c07)    if (seq_q > seq_i) { CH ← CH'; seq_i ←seq_q;}
(c08)    if (CH = φ) { CH ← M; seq_i ←seq_i +1;
(c09)              send (CH, seq_i) to M;}
------------Task c2: Clustering Host ---------------------------------
(c10) clusterhead←i; rejected ←φ; sn ← 0;
      ------Task c2.1: Action of Cluster Member ----------
(c11) while(true){
(c12)    if(clusterhead∉ CH or  (a RELEASE(sn') from
                                m_k with clusterhead = k)){
(c13)          if(clusterhead = i)
                 send RELEASE(sn) to all local hosts except m_i;
(c14)       else
                 send LEAVE(sn) to clusterhead;
(c15)       sn←sn+1; rejected ←φ;
(c16)       if ((CH \rejected)= φ){ rejected ←φ; sn←sn +1;}
(c17)       clusterhead←NEAR(CH \rejected);
(c18)       send JOIN(sn) to clusterhead;
(c19)       wait until ACK(type, sn) received from clusterhead;
(c20)       if(ACK.type = false){
(c21)            rejected ← rejected□{clusterhead};
(c22)            GOTO (c15);}}
      }
      ---------Task c2.2: Action of Clusterhead -------------
(c23) while(true){
(c24)  upon reception of a JOIN(sn) message from a host m_k:
(c25)     if(clusterhead≠i) send ACK(false, sn) to m_k;
(c26)     else {add m_k to local host list;
(c27)          send ACK(true, sn) to m_k;}
(c28)  upon reception of a LEAVE(sn) message from a host m_k:
(c29)     if(clusterhead=i) delete m_k from local host list;
      }
COEND
```

Fig. 1. The implementation of ◊*C*

```
COBEGIN // The code executed by each host, m_i
--------------------Task 1: Consensus ----------------------------------------
(101) r_i ←0; est_i ← v_i; ts_i ← 0; fl_i ← false;
(102) while (fl_i≠true){
(103)    r_i←r_i+1;
(104)    cc = coord(r_i); ch = ◊C.clusterhead;
      ---------- Phase 1: Collecting Proposal ----------------
(105)  if(i=cc) send PROPG(r_i, est_i) to ◊C.CH;
(106)  else send NEWR(r_i) to m_cc;
(107)  if(i=ch){// m_i is a clusterhead;
(108)     wait until (received a PROPG (r_i, v_cc) from m_cc,
                  or m_cc ∉ ◊C.trusted, or m_i∉ ◊C.CH);
(109)     if(PROP(r_i, v_cc)is received) send PROP(r_i, v_cc) to all local hosts
(110)     else send PROP(r_i,⊥) to all local hosts;}
(111)  wait until ((received PROP(r_i, *) from m_ch) or ch ≠ ◊C.clusterhead);
(112)  if(PROP(r_i, v) with v ≠ ⊥ is received) {est_i← v; ts_i=r_i;}
      ----------Phase 2: Collecting Echo --------------------
(113)  send ECHO(r_i, est_i, ts_i) message to m_ch;
(114)  if (i=ch) {
(115)     wait until(received ECHO(r_i,*,*) from
                  each local host m_j∈ ◊C.trusted);
(116)     merge the ECHO messages received into
                  an ECHOG (r_i, estm, tsm, W, Z);
            // tsm: the greatest ts;
            //estm: the estimate value with timestamp tsm;
            //W: the hosts that sent ECHO with tsm;
            //Z: the hosts that sent other ECHO;
(117)     send ECHOG (r_i, estm, tsm, W, Z) to DA;}
(118)  if(m_i∈DA){
(119)     wait until receive ((ECHOG(r_i, *, *, W, Z) with |□W□Z| ≥ n-f) or
                some (ECHOG(-,-,>r_i,-,-) messge;
(120)     est_i← vm, where vm is the highest timestamp tsvm;
(121)     if(tsvm =r_i and ≥ f+1 hosts are included in the W sets with tsvm){
(122)        ∀j ≠ i: send DECISION(est_i) to m_j;
(123)        fl_i←true;}}
      }//endwhile
---------------Task 2:  Reliable Broadcast ----------------------------------------
      upon reception of DECISION(est) from host m_k:
(201) if(fl_i≠true) {
(202)    ∀j ≠ i, k: send DECISION(est) to m_j;
(203)    fl_i←true;}
COEND
```

Fig. 2. The HCD protocol -- Task 1 and Task 2

JOIN, LEAVE, ACK or RELEASE message to avoid the effect of disorder. When a host receives one of such messages, it needs to check the sequence number of the message so that only updated messages are handled.

Based on the properties of ◊*S* and the correctness of the flush algorithm in [7], it is easy to prove the correctness of the proposed implementation of ◊*C*, i.e. the algorithm in Fig. 1 satisfies the properties of ◊*C*. Due to the limit in space, we do not present the proof in this paper.

4. The HCD consensus protocol

HCD adopts the similar message exchange flow as that in the HMR protocol [18], which is a simple and versatile protocol. The system model of HCD is the same as described in Section 3.1 except that an assumption on the number of faulty hosts is added: the maximum number of MHs that can fail in a run of the consensus protocol, denoted as f, is bounded by $n/2$, i.e. $f < n/2$. An MH that crashes in a run is a faulty host, otherwise it is correct.

4.1. Data structures and message types

When executing the HCD protocol, each host m_i needs to maintain some necessary information about its state. Such information is stored in the following variables.

fl_i: the flag indicating whether m_i has made the decision. The initial value is *false*.

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on March 16, 2009 at 00:09 from IEEE Xplore. Restrictions apply.

r_i: the sequence number of the current round that m_i is participating in.

est_i: the current estimate of the decision value. Initially, it is the value proposed by m_i.

ts_i: the timestamp of est_i. The value is the sequence number of the round in which m_i receives the current est_i. The update of ts_i is entailed by the reception of the estimate from a coordinator.

During the execution of the HCD protocol, MHs need to communicate with each other by exchanging messages. The message types involved in a round r_i are as follows.

PROPG(r_i, est_{cc}): the proposal message sent by the coordinator host m_{cc} to all the clusterheads.

PROP(r_i, v): the proposal message sent by a clusterhead to its local hosts. v can be est_{cc} (the estimate value kept by the coordinator host) or "$\perp$"(a value that cannot be decided upon).

ECHO(r_i, est_i, ts_i): the echo message sent by a host m_i to its clusterhead.

ECHOG(r_i, v, tsv, W, Z): the echo message sent by a clusterhead to *Decision_makers* and *Agreement_keepers* (please see Section 4.2 for the definitions of them). An ECHOG message is constructed by merging the ECHO messages sent by the hosts in the cluster. v is the estimate carried by the ECHO message with the highest timestamp and tsv is the timestamp of v. W is the set of MHs that send the ECHO message with the timestamp tsv whereas Z is the set of MHs that send other ECHO messages.

DECISION(*est*): the message sent by an MH to propagate the decision value *est*.

NEWR(r): the message used to lead MHs to a new round r.

4.2. Description of the HCD protocol

The HCD protocol consists of four tasks, which are executed separately and concurrently at each host. Task 1 is the main body of the consensus protocol for making a decision by exchanging messages in rounds. Task 2 is a simple broadcast algorithm for propagating the value decided upon. Task 3 is used to send redeeming messages for handling the late ECHO message or cluster switching, while Task 4 handles futures messages. The pseudocode of Task 1 and Task 2 is shown in Fig. 2, while the pseudocode of Task 3 and Task 4 is shown in Figures 3 and 4 respectively. We describe these tasks in details below.

Task 1: Like most consensus protocols, Task 1 is executed in asynchronous rounds, each of which is divided into two phases. In the beginning of a round r_i, a host m_i first determines the coordinator host using the function *coord*(r_i). *coord*(r) is deterministic, i.e. given the same input, it produces the same result. A simple implementation of *coord*(r) can be *coord*(r) = r *mod* n. Then, m_i queries the clusterer oracle $\Diamond C$ to get the *id* of its clusterhead. The remaining actions during the round are divided into two phases.

In Phase 1, if m_i is the coordinator host, it sends the *PROPG*(r_i, est_i) message to all the clusterheads (line 105); otherwise, m_i sends a *NEWR*(r_i) message to the coordinator host m_{cc}. The *NEWR*(r_i) message is used to wake up the coordinator in case that it is blocked in some previous round due to message losses.

Each clusterhead waits for the PROPG message from m_{cc} unless it is no longer a clusterhead or the coordinator is suspected (line 108). When the *PROPG*(r_i, est_{cc}) message from m_{cc} is received, a clusterhead will forward the message to all its local hosts (line 109); otherwise it sends out the *PROP*(r_i, $\perp$) message to its local hosts (line 110).

Each host m_i waits for a PROP message from its clusterhead m_{ch} unless it switches to another cluster due to mobility or hosts failures (line 111). Upon the reception of the *PROP*(r_i, v) message from its clusterhead, if the estimate value v carried by the PROP message is not equal to $\perp$, m_i updates its estimate est_i to v and timestamp ts_i to r_i (line 112). Phase 1 ends.

Phase 2 is started by sending ECHO messages. Each host m_i first sends an *ECHO*(r_i, est_i, ts_i) message to its clusterhead m_{ch}. If m_i itself is a clusterhead, it waits for the ECHO messages from all its correct local hosts (line 115). m_{ch} first merges all the ECHO messages received into an *ECHOG* (r_i, *estm*, *tsm*, W, Z) message, where *tsm* is the greatest *ts* in the ECHO messages; *estm* is the estimate value with timestamp *tsm*; W is the set of the hosts that have sent the ECHO messages with the timestamp *tsm*; Z is the set of the hosts that have sent other ECHO messages. Then, m_{ch} sends this ECHOG message to all the hosts in a set *DA*.

Let *DA* denote the union set of the set of *Decision_makers* and the set of *Agreement_keepers*. The set of *Decision_makers* contains the hosts that need to check the decision predication to know if they can decide during the current round; the set of *Agreement_keepers* consists of the hosts that should keep the updated estimate value. Here, each host in *DA* simultaneously plays two roles: *Decision_maker* and *Agreement_keeper*. The construction of *DA* must satisfy the following constraints:

i) *DA* is deterministic. *DA* can be changed for different rounds, but during the same round r, all the hosts have the same *DA*.

ii) For each round r, *DA* contains at least the coordinators of round r and round r+1.

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on March 16, 2009 at 00:09 from IEEE Xplore. Restrictions apply.

The hosts in *DA* wait for the *ECHOG*(r_i, *, *, *W*, *Z*) messages until: i) the ECHOG messages received include at least *n-f* hosts, i.e. $|\cup W \cup Z| \geq n\text{-}f$, or ii) an *ECHOG*(-,-,$ts_v$,-,-) message with $ts_v > r_i$ is received (line 119). Then, each host m_{da} in the *DA* set updates its estimate to the value carried by the ECHOG message with the highest timestamp *tsvm*. If $tsvm = r_i$ and no less than f+1 hosts are included in the *W* sets of all the messages with the timestamp *tsvm*, m_{da} makes the decision upon *vm* and sends *vm* to all the other hosts using the *DECISION*(*vm*) message. Phase 2 ends.

Task 2: This task simply broadcasts the decision value. When a host receives a DECISION message, it decides upon the value carried by the DECISION message and forwards the message to all the other hosts except the sender.

Task 4: This task handles future messages. A message *msg* is a future message if it arrives at an MH before the MH enters the corresponding round of *msg*, i.e. the round in which *msg* is sent. When an MH m_i receives a future message with a round number $r > r_i$, it will jump to a future round after performing the following operations:

i) If m_i is a clusterhead host and *msg* is not a NEWR message, m_i sends the *NEWR*(*r*) message to its local hosts, so that the local hosts can also jump to a future round.
ii) If m_i is not in the *DA* set of round *r*-1, it will jump to round *r* (line 406). Before the jump, m_i sends the ECHO messages for the rounds skipped (line 405).

```
------------Task 3: Sending Redeeming Messages----------
// The code executed by each host m_j;

(301) upon reception of ECHO(r_i, *, *) from m_i, with ((r_i<r_j) or
          (r_i=r_j but m_j has sent an ECHOG(r_i, *, *, *, *))){
(302)     construct an ECHOG for the ECHO and send it to DA of round r_i;
      }
(303) upon the change of local clusterhead{
(304)     if(m_j has not entered Phase 2)
              for(ts_j≤ rr< r_j) send ECHO(rr, est_j, ts_j) to the new clusterhead;
(305)     else
              for(ts_j≤ rr≤ r_j) send ECHO(rr, est_j, ts_j) to the new clusterhead;
      }
```

Fig. 3. The HCD protocol -- Task 3

```
------------Task 4: Handling Future Messages-------------
// The code executed by each MH m_i;

(401) upon reception of a message msg with r>r_i: {
(402)     if(i=ch and msg is not a NEWR message)
(403)         send NEWR(r_i) to local hosts;
(404)     if(m_i is not in the DA set of round r-1) {
(405)         for(r_i<rr<r) send ECHO(rr, est_i, ts_i) to m_ch;
(406)         r_i← r; GOTO (104);
(407)     }else if(r-1>r_i){
(408)         for(r_i<rr<r-1) send ECHO(rr, est_i, ts_i) to m_ch;
(409)         r_i← r-1; GOTO (104);}
      }
```

Fig. 4. The HCD protocol -- Task 4

Task 3: This task sends redeeming messages caused by a late ECHO message or cluster switching. An ECHO message is "late" if it arrives at a host after this host has sent out an ECHOG message (line 117) for the corresponding round. This happens when a clusterhead m_{ch} suspects a correct cluster member or a host newly joins the cluster. The *Decision_makers* or *Agreement_keepers* may be blocked forever if an ECHO message is ignored. To avoid this, when a host m_j receives an ECHO message from a host m_i for round r_i where ($r_i < r_j$) or ($r_i = r_j$ but m_j has sent out an ECHOG message for round r_i), m_j constructs a redeeming ECHOG message and sends it to the *Decision_makers* or *Agreement_keepers* of round r_i.

A host m_j may need to change its cluster during the execution. Its previous clusterhead may have crashed and lost the messages forwarded for m_j. Therefore, upon the change of its clusterhead, m_j needs to send some redeeming messages (line 303). m_j resends the ECHO messages to the new clusterhead for the rounds between ts_j and r_j. For round r_j, the current round of m_j, if m_j has sent out an *ECHO*(r_j, est_j, ts_j) message, it resends this message to the new clusterhead.

iii) Otherwise, if m_i is in the *DA* set of round *r*-1 (it needs to execute line 119 of round *r*-1) and round *r*-1 is also a future round, it will jump to round *r*-1 (line 409). Before the jump, m_i sends the ECHO messages for the rounds skipped (line 408).

The difference in the operations for Case ii) and Case iii) is necessary for guaranteeing the agreement property.

We have proved that the HCD protocol can guarantee all the correctness properties, i.e. termination, agreement, and validity. However, due to the limit in space, we do not present the correctness proof of the HCD protocol in this paper.

5. Performance evaluation

In this section, we analyze the message cost of the proposed HCD protocol in comparison with the HMR protocol [18] and our previous HCS protocol.

5.1. Performance metrics

In traditional distributed systems, the message cost is computed in terms of the number of end-to-end

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on March 16, 2009 at 00:09 from IEEE Xplore. Restrictions apply.

messages. However, one message may take one or more hops to reach the destination. One "hop" means one network layer message, i.e. a point-to-point message. In traditional systems, the costs of messages transmitted in different numbers of hops are viewed as the same. In a MANET, however, the resource is seriously constrained, so the cost must be measured more precisely in hops.

Due to the asynchrony of the consensus protocol execution, it is almost impossible to evaluate the total number of rounds in achieving consensus by numerical simulations. Therefore, we analyze only the message cost per round. However, we expect that HCD can achieve a larger improvement in the overall performance than that in the performance per round. As shown by the simulation results reported in [29], a hierarchical consensus protocol may achieve consensus using fewer rounds than existing solutions, e.g. HMR, which is the gain obtained by the synchronization between cluster members and the clusterhead.

The following two metrics are used in the following evaluations:

NMR (*Number of Messages per Round*): the total number of messages exchanged in a round.

NHR (*Number of Hops per Round*): the total number of hops of the messages exchanged in a round.

Alothough, the HCS protocol adopts the FD of $\Diamond P$, which is stronger than $\Diamond S$, the FD used by HMR and HCD, the properties of the FD only affect the number of rounds needed to achieve consensus rather than the message cost of one round. Therefore, the difference in FD does not affect the performance evaluation here.

5.2. Evaluations and comparisons

5.2.1. Message cost of HMR. As discussed in [29], the HMR protocol performs nearly the best when only the coordinators of the current and next rounds act as the *Decision_makers* and *Agreement_keepers*. Therefore, in the following analysis, we assume that there are only two hosts in the *DA* set in each round. In Phase 1, the coordinator sends a PROP message to each host. In Phase 2, each host sends two ECHO messages to the *Decision_makers* and *Agreement_keepers*. Thus, we have:

$$NMR_{hmr} = n + 2n = 3n \quad (1)$$

Since the topology of a MANET can be represented by a graph, the average number of hops of an end-to-end message is related to the diameter of the graph. We adopt the value log^n [26] as the average number of hops of an end-to-end message. Then, we have:

$$NHR_{hmr} = 3n \bullet log^n \quad (2)$$

5.2.2. Message cost of HCS. In the HCS protocol, the clusterhead set is static and contains of k hosts. In Phase 1, the coordinator sends the PROP message to each clusterhead and then, the clusterheads forward the PROP message to each local host. Obviously, $n+k$ messages are exchanged in this phase.

In Phase 2, each host sends an ECHO message to its clusterhead and the clusterhead merges all the ECHO messages received into one ECHOG message and sends it to the *Decision_makers* and *Agreement_keepers*. Same as in HMR, only the coordinators of the current and next rounds act as the *Decision_makers* and *Agreement_keepers*, so the number of messages exchanged in Phase 2 is $n+2k$. Then, we have:

$$NMR_{hcs} = (k+n) + (n+2k) = 2n + 3k$$

NHR_{hcs} depends on the distance between the cluster members and their clusterheads. Let l be the average number of hops of one message between a host and its clusterhead. In Phase 1, the number of hops is $k \bullet log^n + n \bullet l$; in Phase 2, the number of hops is $n \bullet l + 2k \bullet log^n$. Then, we have:

$$NHR_{hcs} = 2n \bullet l + 3k \bullet log^n$$

Since an MH always attempts to choose the nearest clusterhead, each cluster can be viewed as a sub-network with the number n/k of hosts. Therefore, $l = log^{(n/k)}$. Then, we have:

$$\begin{aligned} NHR_{hcs} &= 2n \bullet log^{(n/k)} + 3k \bullet log^n \\ &= (2n+3k) \bullet log^n - 2n \bullet log^k \end{aligned}$$

As discussed in [29], the number of clusterhead hosts k is the key parameter for hierarchical protocols. In HCS, to guarantee the same capability of fault tolerance as the HMR protocol (i.e. at most there can be $n/2$-1 faulty hosts), k is set to $n/2$. Then, we have:

$$NMR_{hcs} = 2n + 3k = 7n/2 \quad (3)$$

$$NHR_{hcs} = 2n + (3/2)n \bullet log^n \quad (4)$$

5.2.3. Message cost of HCD. In the HCD protocol, we also denote the number of clusterheads by k. In Phase 1, the coordinator host sends the PROPG message to all the clusterheads and the clusterhead sends the PROP message to each cluster member. The number of messages in Phase 1 is $k+n$. The message exchange pattern in Phase 2 of HCD is the same as in HCS, so the number of messages in Phase 2 is also $n+2k$. Then, we have:

$$NMR_{hcd} = (k+n) + (n+2k) = 2n + 3k$$

Following the same approach of computing the number of hops per message used above, we have:

$$\begin{aligned} NHR_{hcd} &= (k \bullet log^n + n \bullet l) + (n \bullet l + 2k \bullet log^n) \\ &= 2n \bullet log^{(n/k)} + 3k \bullet log^n \\ &= (2n+3k) \bullet log^n - 2n \bullet log^k \end{aligned}$$

Same as in HCS, the size of the clusterhead set is an important factor that affects the performance of HCD. Different from HCS, the clusterhead set in HCD is dynamically changed by the clusterer oracle. In the description of the HCD protocol, for simplicity, the clusterhead set is initialized to M, which is obvious not

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on March 16, 2009 at 00:09 from IEEE Xplore. Restrictions apply.

efficient in terms of message cost. Here, we assume that, initially one half of all the hosts are put in the clusterhead set. Therefore, we use $n/2$ as the upper bound of the clusterhead set size. The lower bound can be one, because the FD $\Diamond S$ is used and, finally, there may be only one host in the clusterhead set. Roughly, we have the average number of clusterhead $k \approx (n/2 + 1)/2 = (n+2)/4$. Then, we have:

$$NMR_{hcd} = 2n + 3k = 11n/4 + 3/2 \quad (5)$$

$$NHR_{hcd} = (2n+3k) \bullet log^{n} - (2n+1) \bullet log^{k} = (11n/4+3/2) \bullet log^{n} - (2n+1) \bullet log^{(n+2)} + 4n + 2 \quad (6)$$

5.2.4. Numerical results and discussions. Using the equations (1) – (6), we computed the results in *NMR* and *NHR*, which are plotted in Figures 5 and 6 respectively. In terms of *NMR*, HCD performs the best while HCS performs the worst. The difference between HMR and the other two protocols comes from the message forwarding mechanism introduced by the cluster-based hierarchy. In Phase 1, HMR needs the fewest messages but in Phase 2 the difference is in fact determined by k, the size of the clusterhead set. In Fig. 5, the k of HCS is equal to $n/2$ while the k of HCD is $(n+2)/4$. This causes that the performance of HMR in *NMR* is in the middle.

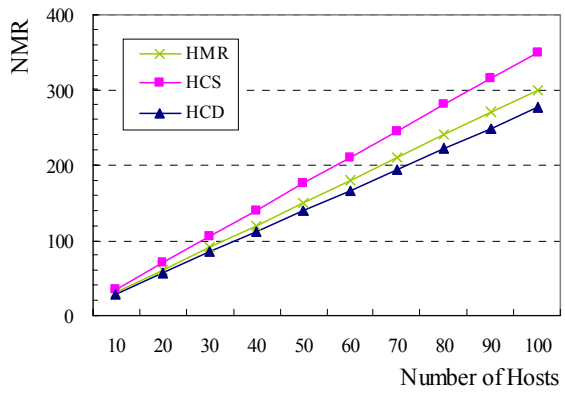

Fig. 5. Message cost in *NMR*

As discussed in Section 5.1, *NHR*, the message cost in number of hops, is more precise and useful than *NMR*. Fig. 6 shows that HCD still performs the best and HCS can also significantly reduce the message cost of achieving consensus. Such benefit is gained by the two-layer hierarchy. With the increase of the number of hosts, the benefit also increases. Therefore, the hierarchical protocols have better scalability than HMR. Moreover, due to the smaller k in HCD than that in HCS, HCD can perform better than HCS. Of course, this does not mean that the smaller the k is the better the protocol performs. As discussed in [29], when k is in the middle (about $n/3$), the protocol can perform the best.

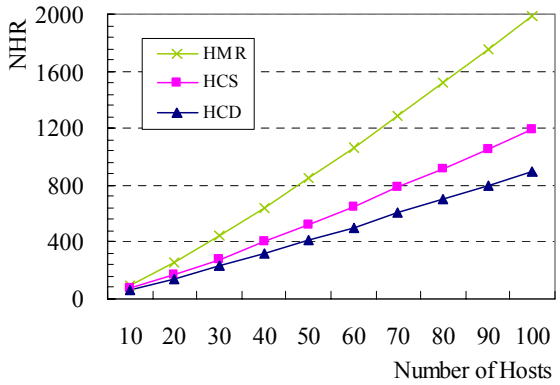

Fig. 6. Message cost in *NHR*

6. Conclusions and future work

In this paper, we have proposed a modular approach to the design of message efficient hierarchical consensus protocols for MANETs, which uses a cluster-based two-layer hierarchy imposed on the mobile hosts to reduce message cost. The modular approach separates clustering from achieving consensus by defining a new oracle, namely the eventual clusterer $\Diamond C$. $\Diamond C$ can construct a cluster-based hierarchy over and detect the failures of the mobile hosts. Different from the existing work, the clusterhead hosts, which form the upper layer of the hierarchy, are dynamically selected by $\Diamond C$, so the protocol can adapt the hierarchy to the failure of hosts. $\Diamond C$ is a separate module transparent to the consensus protocols built on top of it, so it can be used to design different message efficient consensus protocols.

Based on $\Diamond C$, we have designed a hierarchical consensus protocol, HCD, where the messages from and to the hosts in the same cluster are merged/unmerged by the clusterhead so as to reduce the message cost and improve the scalability. An implementation of $\Diamond C$ is also presented in the paper, which is based on the weakest unreliable failure detector $\Diamond S$.

The numerical analysis results show that the proposed protocol can save much message cost compared with the existing work. Due to the limit of the numerical analysis, the overhead of clustering is not taken into consideration in the performance evaluation. In the future, we will evaluate the performance of the proposed consensus protocol by conducting simulations. The message cost of a complete execution of our proposed protocol will be measured and the overhead of the clustering function will also be examined.

Acknowledgements. This research is partially supported

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on March 16, 2009 at 00:09 from IEEE Xplore. Restrictions apply.

by the University Grant Council of Hong Kong under the CERG Grant PolyU 5105/05E, PROCORE France-Hong Kong grant F-HK16/05T, and the China National "973" Program Grant 2007CB307100.

References

[1] N. Badache, M. Hurfin, and R. Macedo, Solving the Consensus Problem in a Mobile Environment, *Proc. of IPCCC'99*, 1999.

[2] B. Badrinath, A. Acharya, and T. Imielinski, Designing distributed algorithms for mobile computing networks, *Computer Communications*, vol. 19, no. 4, Apr. 1996.

[3] M. Ben-Or, Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols, *Proc. of PODC '83*, 1983.

[4] T. Chandra, V. Hadzilacos, and S. Toueg, The Weakest Failure Detector for Solving Consensus, *Journal of the ACM*, vol. 43, no. 4, Jul. 1996.

[5] T. Chandra and S. Toueg, Unreliable Failure Detectors for reliable distributed Systems, *Journal of the ACM*, vol. 43, no. 2, Mar. 1996.

[6] G. Chockler, M. Demirbas, S. Gilbert, C. C. Newport, and T. Nolte, Consensus and Collision Detectors in Wireless Ad Hoc Networks, *Proc. of PODC'05*, Jul. 2005.

[7] F. Chu, Reducing Ω to ◊*W*, *Information Processing Letters*, vol. 67, no. 6, 1998

[8] G. Coulouris, J. Dollimore, and T. Kindberg, Distributed Systems: Concepts and Design (3rd edition), *Addison-Wesley*, 2001.

[9] D. Dolev, R. Friedman, I. Keidar, and D. Malkhi, Failure Detectors in Omission Failure Environments, *Technical Report: TR96-1608*, Department of Computer Science, Cornell University, Sep. 1996.

[10] P. Dutta, and R. Guerraoui, Fast Indulgent Consensus with Zero Degradation, *EDCC'02, LNCS 2485*, 2002.

[11] H. Elaarag, Improving TCP Performance Over Mobile Networks, *ACM CSUR*, vol. 34, no. 3, Sep. 2002.

[12] P. Ezhilchelvan, A. Mostefaoui, and M. Raynal, Randomized Multivalued Consensus, *Proc. of the 4th IEEE International Symposium on Object-Oriented Real-Time Computing*, May 2001.

[13] M. Fischer, N. Lynch, and M. Paterson, Impossibility of Distributed Consensus with One Faulty Process, *Journal of the ACM*, vol. 32, no. 2, Apr. 1985.

[14] G. Forman, and J. Zahorjan,The Challenges of Mobile Computing, *IEEE Computer*, vol. 27, no. 4 1994.

[15] R. Guerraoui, M. Huifin, A. Mostefaoui, R. Oliveira, M. Raynal, and A. Schiper, Consensus in Asynchronous Distributed Systems: A Concise Guided Tour, *LNCS 1752*, 2000.

[16] R. Guerraoui, and A. Schiper, Consensus: the Big Misunderstanding, *the Sixth IEEE Workshop on Future Trends of Distributed Computing Systems*, 1997.

[17] R. Gueraoui, A. Schiper, The Generic Consensus Service, *IEEE Transactions on Software Engineering*, vol. 27, no. 1, Jan. 2001.

[18] M. Hurfin, A. Mostefaoui, and M. Raynal, A Versatile Family of Consensus Protocols Based on Chandra-Toueg's Unreliable Failure Detectors, *IEEE Trans. on Computers*, vol. 51, no. 4, Apr. 2002.

[19] M. Hurfin, and M. Raynal, A Simple and Fast Asynchronous Consensus Protocol Based on a Weak Failure Detector, *Distributed Computing*, vol. 12, no. 4, Sep. 1999.

[20] L. Lamport, The Part-Time Parliament, *ACM Trans. on Computer Systems*, vol. 16, no. 2, 1998.

[21] N. Lynch, Distributed Algorithms, *Morgan Kaufmann*, 1996.

[22] A. Mostefaoui, and M. Raynal, Leader-based Consensus, *Parallel Processing Letters*, vol. 11, no. 1, 2001.

[23] M. Satyanarayanan, Fundamental Challenges in Mobile Computing, *Proc. of PODC'96,* 1996.

[24] A. Schiper, Early Consensus in an Asynchronous System with a Weak Failure Detector, *Distributed Computing*, vol. 10, no. 3, Oct. 1997.

[25] H. Seba, N. Badache, and A. Bouabdallah, Solving the Consensus Problem in a Dynamic Group: an Approach Suitable for a Mobile Environment, *Proc. of ISCC'02*, 2002.

[26] Mukesh Singhal, A Taxonomy of Distributed Mutual Exclusion, *Journal of Parallel and Distributed Computing,* 18, 1993.

[27] K. Sundaresan, V. Anantharaman, H. Hsieh, and R. Sivakumar, ATP: a Reliable Transport Protocol for Ad-hoc Networks, *Proc. of ACM MobiHoc'03*, Jun. 2003.

[28] E. Vollset, and P. D. Ezhilchelvan, Design and Performance-Study of Crash-Tolerant Protocols for Broadcasting and Reaching Consensus in MANETs, *Proc. of SRDS'05*, Oct. 2005.

[29] W. Wu, J. Cao, J. Yang, and M. Raynal, Design and Performance Evaluation of Efficient Consensus Protocols for Mobile Ad Hoc Networks, *IEEE Transactions on Computers*, vol. 56, no. 8, Aug. 2007.

[30] X. Yu, Improving TCP Performance over Mobile Ad Hoc Networks by Exploiting Cross-layer Information Awareness, *Proc. of MobiCom'04*, Sep. 2004.

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on March 16, 2009 at 00:09 from IEEE Xplore. Restrictions apply.